Heaps

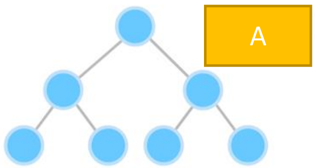
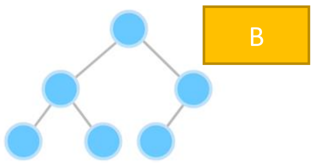
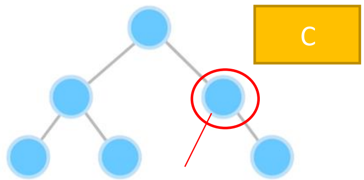
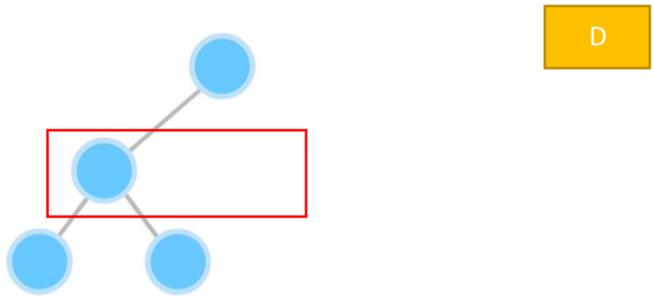
V-(1+2)-What are Heaps:-

Heaps/Binary Heaps are special kind of tree with ***two*** properties.

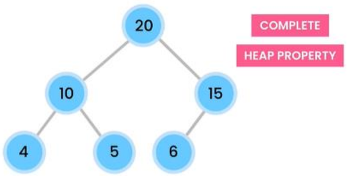
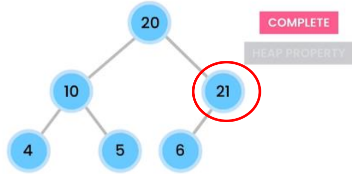
[N.B- (1) In a max **Heap,** root Node has always contains the **largest** value.

(2) In a min **Heap,** root Node has always contains the ***smallest*** value.]

1. It must be a complete tree. (Tree is full with no nodes missing from left to right and also it fills from level to level (that's how node will be inserted for making a tree complete.))

<table>
<tr>
<td>It is complete because every level except the last levels are full with nodes from left to right.</td>
<td>It is also a complete tree because levels are full from left to right</td>
<td>This is not at all a complete tree because nodes are not filling from left to right properly. Carefully looked at the red circle without even filling it left child the node fills it right child which completely against a complete tree.</td>
</tr>
<tr>
<td colspan="3">It is not a complete tree because 2nd level is not fulled.</td>
</tr>
</table>

2. All the parent nodes value is greater or equal to its children value. That's called heap property. [In short:- **parent ≥ child**].

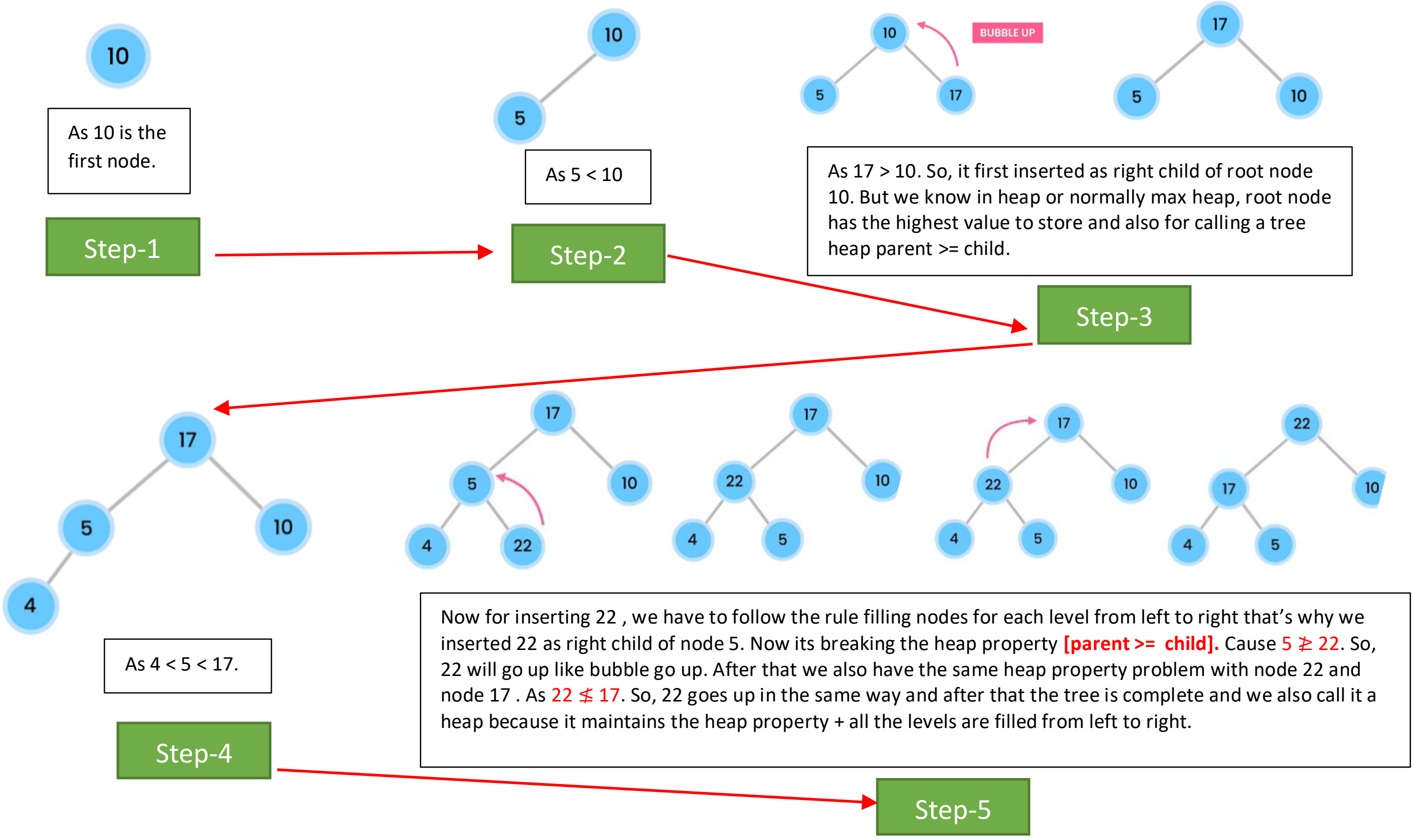
 It is a complete tree cause levels are manage filling from left to right and also every parent nodes value is greater than its equivalent child values (heap property).	 Though it is a complete tree but one child value is greater than its parent value that means heaps property is missing thus this tree is not a heap. (21>20)

Heaps Uses:-

- Sorting (HeapSort)
- Graph algorithms (shortest path)
- Priority queues
- Finding the Kth smallest/largest value

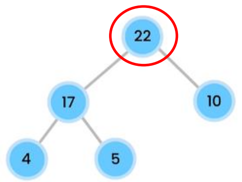
Heaps Operations:-

Lets make a heap (**Insertion**).[10, 5, 17, 4, 22]



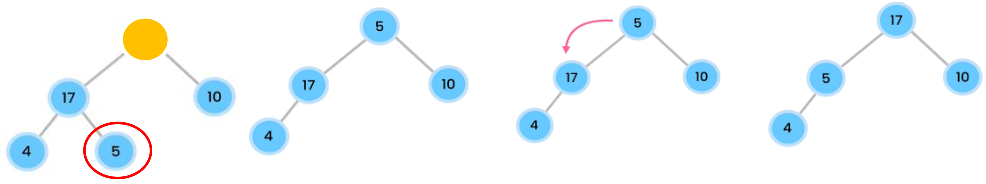
Time complexity of **insertion** in a **heap** would be **O(log(n))**. Cause it is like in **BST (Binary Search Tree)** for finding an element we have to **traverse maximum the height of a tree**. In heap we also do the same but comparison are in opposite direction (as we are comparing child with parents and then goes up accordingly following heap property).

Deletion of nodes :- (In heap we only delete root node normally.)



22 will be deleted.

Step-1



Now root node is empty. So, first check our tree remains complete or not. For making a tree complete nodes must be inserted from left to right for each level. So, 5 goes as root. But it breaks the heap property. So, as we saw bubbling up but in this case we see bubbling down that means 5 needs to go down. Now, we have to check the children value of root node 5 and compare which is the largest one among them (17, 10) cause largest value will be existed in the root node as it is max heap. As 17 > 10, we choose 17 to be root and 5 goes down and also after that we check 5 is greater than or equals to its child or not and as 5 > 4. Now our tree is both complete and it maintains the heap property.

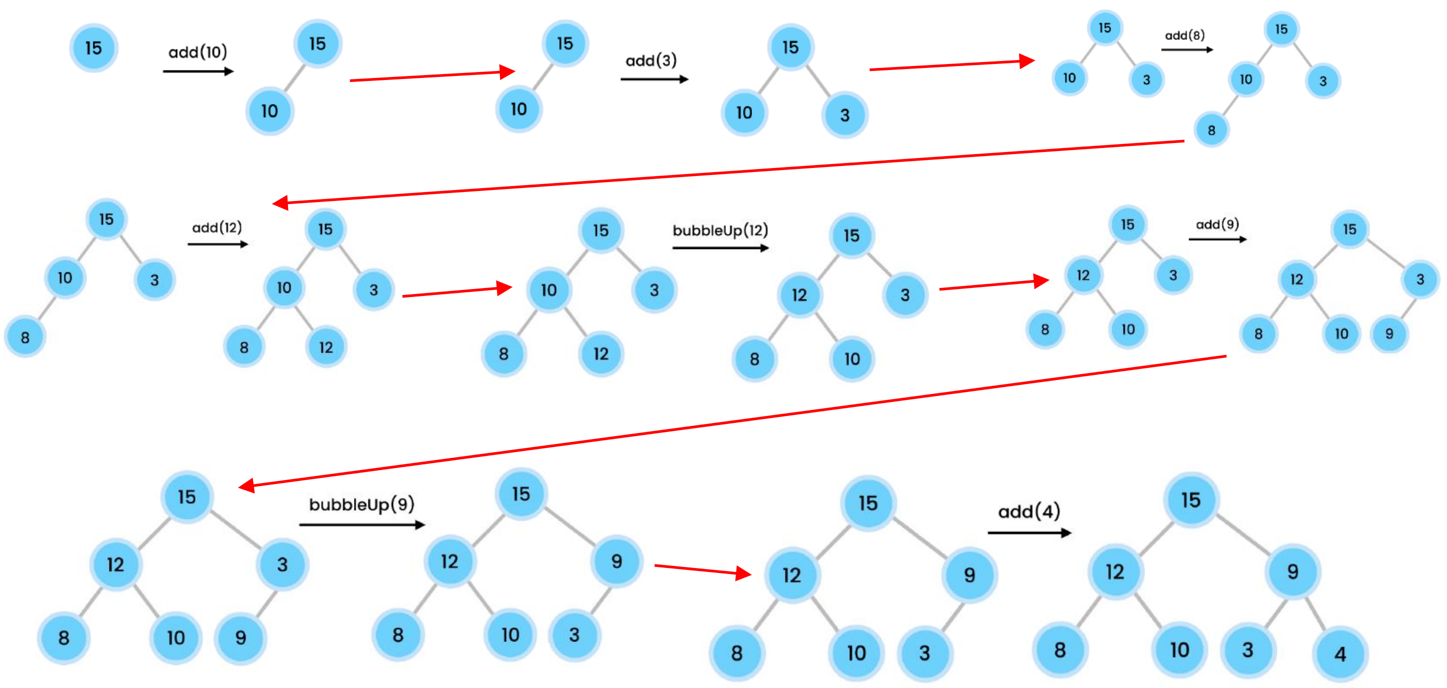
Step-2

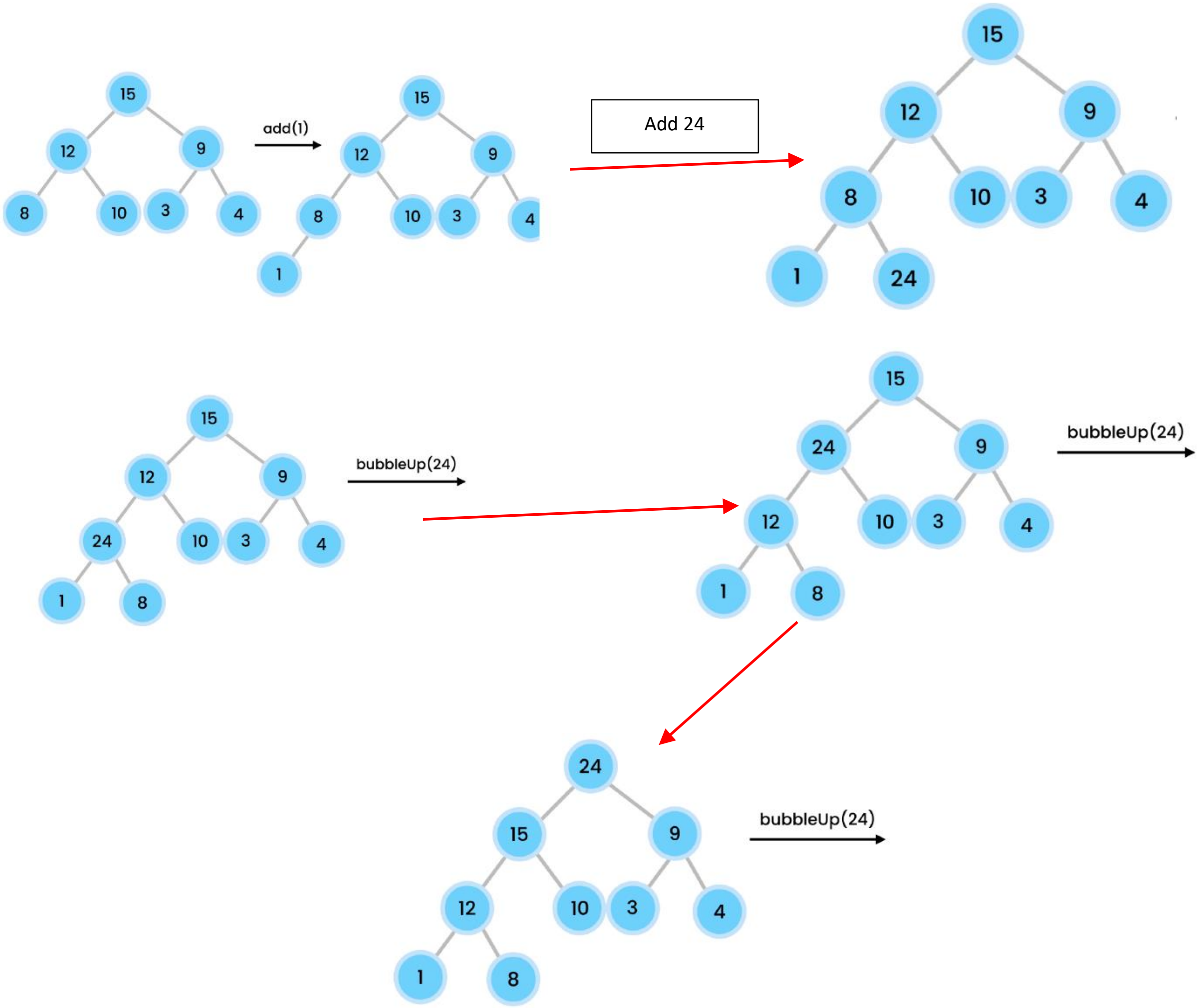
In **max heap**, as root node consists the highest value so for finding highest value from max heap we simply return the root. So in this case time complexity will be O (1).

In **min heap**, as root node consists the smallest value so for finding smallest value from min heap we simply return the root. So in this case time complexity will be O (1).

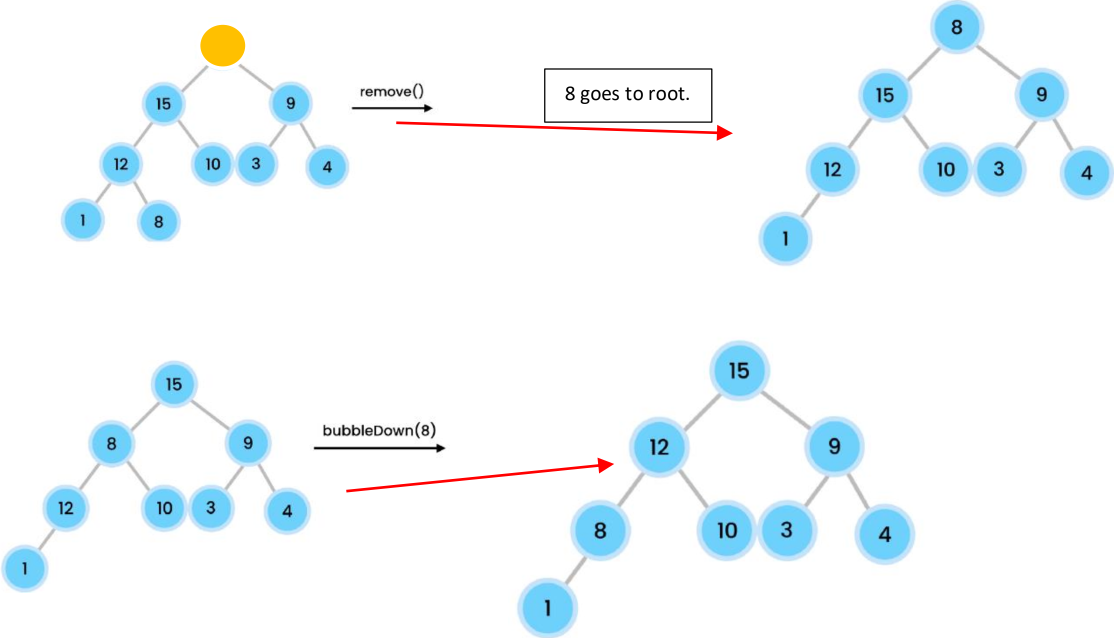
V-(3)- Heaps Exercise:-

Q-1) Insert the following numbers in a heap. Draw the heap at each step.
Compare your solution with mine in the following pages.
(15, 10, 3, 8, 12, 9, 4, 1, 24)

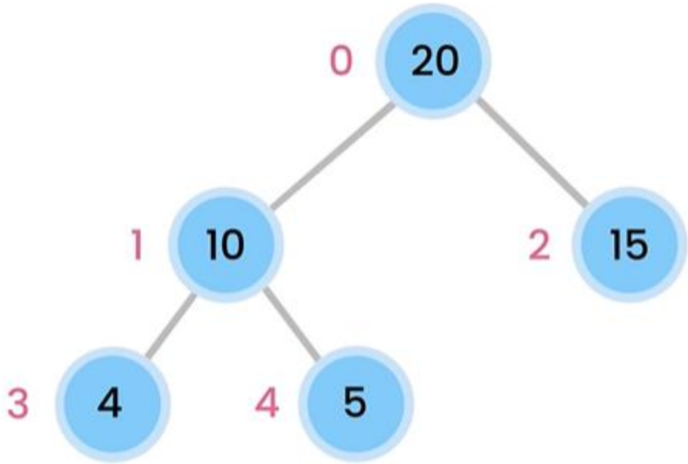




Q-2) remove the root (24) from the heap.



V-(4+5)- Building a Heap(insert method):-



left = parent * 2 + 1

right = parent * 2 + 2

parent = (index - 1) / 2

```
public class Heap {
    private int items[] = new int[10];
    private int size;
    public void insert(int value){
        if(isFull())
            throw new IllegalStateException();
        items[size++] = value;
        bubbleUp();
    }
    private boolean isFull(){
        return size==items.length;
    }
    private void bubbleUp(){
        var index = size-1;
        while(index>0 && items[index]>items[parent(index)]){
            //bubble up
            swap(index,parent(index));
            //    10  =====>>>   10      =====>>> 12 (i)
            //   5 ========>>> 12 (i) ======>> 10
            // 12 i =======> 5   ==========>> 5
            // as everytime index will be changed so, index will become parent
index
            index = parent(index);
        }
    }
    private int parent(int index){
        return (index-1)/2;
    }
    private void swap(int first,int second){
        var temp = items[first];
        items[first]=items[second];
        items[second]=temp;
    }
}
```

```
//time complexity -> O(log n)
// inserting item in the leaf
//then if it is not in right position
//we must have to bubble up for fixing it
//in that case in worst case item has to
travel the
//height of a tree which is O(log n).
// if we have too many insertion we can
use heap to insert cause it is faster.
```

V-(6)- Remove method-(considersing every level is full):-

```
private void swap(int first,int second){
    var temp = items[first];
    items[first]=items[second];
    items[second]=temp;
}
public void remove(){
    //first check if the heap is empty or not
    if(isEmpty())
        throw new IllegalStateException();
    //for removing we remove the last item and replace it with root ,
    // then set it root and bubble down if needed for heap property
    items[0]=items[--size];//first decrement the size to get the last
//item and insert it in root
    //now its time for bubble down if (root<leftchild or
root>rightchild)
    var index = 0;
    while(index<=size/* at some point index may be -ve then we
stopped*/ && !isValidParent(index)){
        //find which one is larger (left one or right one)
        var largerChildIndex = largerChildIndex(index);
        swap(index,largerChildIndex);
        index = largerChildIndex;//index can be changed multiple times
before going to its right position
    }
}
private int largerChildIndex(int index){
    return (leftChild(index)>rightChild(index))?
            leftChildIndex(index):rightChildIndex(index);
}
private boolean isValidParent(int index){
    return items[index]>=leftChild(index) &&
items[index]>=rightChild(index);
}
private int rightChild(int index){
    return rightChildIndex(index);
}
private int leftChild(int index){
    return leftChildIndex(index);
}
private int leftChildIndex(int index){
    return (index*2)+1;
}
private int rightChildIndex(int index){
    return (index*2)+2;
}
public boolean isEmpty(){
    return size==0;
}
```

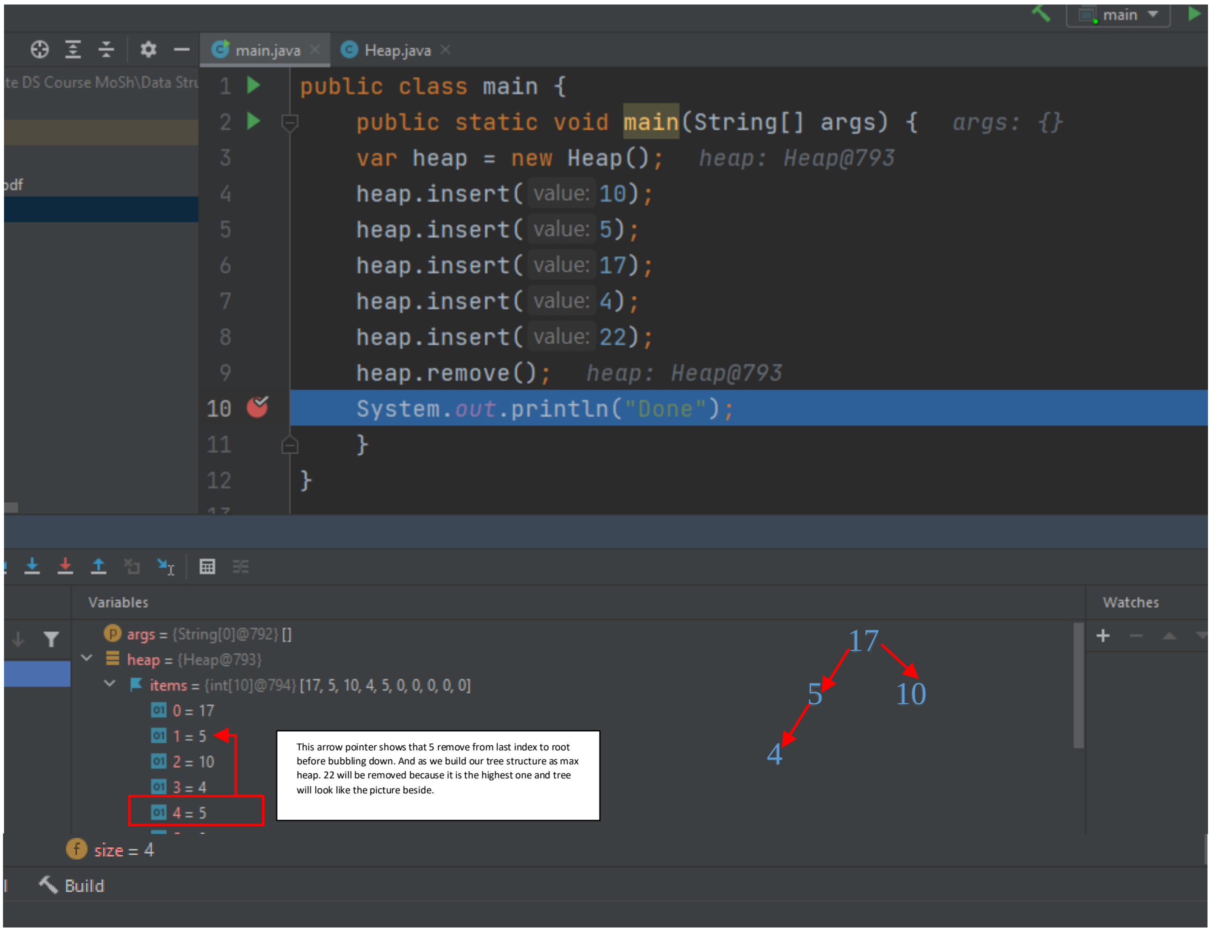
V-(7)- Remove method-(Edge Cases (resolving isValidParent and LargerChildIndex Method)):-

How to know if a node has leftChild Or rightChild. Its very simple compare the index of leftChild or rightChild with size. If it is <= size then it is a valid index.

```
private void swap(int first,int second){
    var temp = items[first];
    items[first]=items[second];
    items[second]=temp;
}
public int remove(){//slower delete
    //first check if the heap is empty or not
    if(isEmpty())
        throw new IllegalStateException();
    //for removing we remove the last item and replace it with
root ,
    // then set it root and bubble down if needed for heap
property
    var root = items[0];
    items[0]=items[--size];//first decrement the size to get the
last item and insert it in root
    //now its time for bubble down if (root<leftchild or
root>rightchild)
    bubbleDown();
    return root;
}
private void bubbleDown(){
    var index = 0;
    while(index<=size/* at some point index may be -ve then we
stopped*/ && !isValidParent(index)){
        //find which one is larger (left one or right one)
        var largerChildIndex = largerChildIndex(index);
        swap(index,largerChildIndex);
        index = largerChildIndex;//index can be changed multiple
times before going to its right position
    }
}
private boolean hasLeftChild(int index){
    return leftChildIndex(index)<=size;
}
private boolean hasRightChild(int index){
    return rightChildIndex(index)<=size;
}
private int largerChildIndex(int index){
    //Now first we check if a node has a left child or not
    //if we don't have any leftchild that means this node doesn't
have any children
    //because we fill nodes from left to right
    if(!hasLeftChild(index))
        return index;//leftChild na thakle oi index ei return
korbo
    //now we see if the node has any rightChild or not if not
```

```
//O(log n) cause in worst
cases it has to traverse
the height of the tree
which is O(log n).
```

```
then
    // we should return the leftChild Index cause it has only
leftChild left
    if(!hasRightChild(index))
        return leftChildIndex(index);
    //now the 3rd case is the node both having its left and
rightchild
    //in that case we have to compare between them and find the
index with
    //larger value
    return
(leftChild(index)>rightChild(index))?leftChildIndex(index):rightC
hildIndex(index);
}
private boolean isValidParent(int index){
    //if a node doesn't have any leftChild that means it is valid
    if(!hasLeftChild(index))
        return true;
    //now if the condition shown above is false that means we do
have leftchild
    // then we compare the leftChild value with its parents
    var isValid = items[index]>=leftChild(index);
    //now if we also have right child then we have to think of
both of the children of that node
    if(hasRightChild(index))
        isValid &= items[index]>=rightChild(index);
    //now 3rd case we check for both children with its parent
that
    //parent must be greater than both of the children
    return isValid;
}
private int rightChild(int index){
    return items[rightChildIndex(index)];
}
private int leftChild(int index){
    return items[leftChildIndex(index)];
}
private int leftChildIndex(int index){
    return (index*2)+1;
}
private int rightChildIndex(int index){
    return (index*2)+2;
}
public boolean isEmpty(){
    return size==0;
}
```



V-(8)- Heap Sort:-

```
var heap2 = new Heap();
int []numbers = {5,3,10,1,4,2};
heap.insert(10);
heap.insert(5);
heap.insert(17);
heap.insert(4);
heap.insert(22);
for(int i=0;i< numbers.length;i++){//inserting items in heap2 object
which is also a max heap
    heap2.insert(numbers[i]);
}
/*while(!heap2.isEmpty())
    System.out.println(heap.remove());
    // as we build our heap as max heap so if we remove items from heap
    //all items will remove in descending order.
 */
//if we want our items to be removed in ascending order
// we can use for loop and backward iteration
for(int i= numbers.length-1;i>= 0;i--){
    numbers[i]= heap2.remove();
}
   for(var number:numbers)
       System.out.println(number+" ");
//System.out.println(Arrays.toString(numbers));
}
```

V-(9)- Priority Queue+Heap:-

```
import java.util.Arrays;

public class PriorityQueue {
    private int[] items =  new int[5];
    private int count;

    public void add(int item){
```

```
//O(n)-> cause in worst cases we have to shift all items to its
rights.
```

```
        if(isFull())
            throw new IllegalStateException();

        var i = shiftItemsToInsert(item);
        items[i] = item;
        count++;
    }
    public boolean isFull(){
        return count== items.length;
    }
    public int shiftItemsToInsert(int item){
        int i;
        for(i=count-1;i>=0;i--){
            if(items[i]>item)
                items[i+1]=items[i];//shifting items
            else
                break;
        }
        return i+1;
    }
    public boolean isEmpty(){
        return count==0;
    }
    public int remove(){// considering highest number get the higher
priority
        if(isEmpty())
            throw new IllegalStateException();
        return items[--count];
    }
    @Override
    public String toString(){
        return Arrays.toString(items);
    }
}
```

new class which is a wrapper of heap class:-

```
public class PriorityQueueWithHeap {
    private Heap heap = new Heap();
    //this is simply a wrapper class of heap.
    public void enqueue(int item){//O(log n)
        heap.insert(item);
    }
    public int dequeue(){//O(log n)
        return heap.remove();
    }
    public boolean isEmpty(){
        return heap.isEmpty();
    }
}
```

V-(10+11)- Heapify:-

```
public class Main {
    public static void main(String[] args) {
        int[] numbers = { 5, 3, 8, 4, 1, 2 };
        // heapify(array)
    }
}
```

We just have to design a method which takes array and create a heap. In our case we follow max heap . So we just check that every parent is greater or equals to its child or not. If the rules violate then we just swap the items with bubble down recursively (inplace heapify).

```
public class MaxHeap {
    public static void heapify(int array[]){
    for (var i = 0;i<array.length;i++)//now in each iteration we will heapify
        heapify(array,i);
    }
    private static void heapify(int array[],int index){
    var largerIndex = index;
        //we first assuming this(root is larger) index is okay and
        // the value of the index is greater or equals to its left & right
child
        // but we have to validate this by using below steps.
        var leftIndex = index * 2 +1;
        //Now, if left child is greater than its parent
        // then we have to swap the reference of both the indexes
        if(leftIndex<array.length && array[leftIndex]>array[largerIndex])
            largerIndex = leftIndex;
        //Now, if right child is greater than its parent
        // then we have to swap the reference of both the indexes
        var rightIndex = index * 2 +2;
        if(rightIndex<array.length && array[rightIndex]>array[largerIndex])
            largerIndex = rightIndex;
        //if the item is greater or equal with both of its children
        //then we just have to return; that means we don't have to do anything
```

```
        if(index==largerIndex)//base case
            return;
        //Other than the cases we have to swap items
        swap(array,index,largerIndex);
        heapify(array,largerIndex);
        //for the branches we have to do it recursively
    }
    private static void swap(int[]array, int first, int second){
        var temp = array[first];
        array[first]=array[second];
        array[second]=temp;
    }
}
```

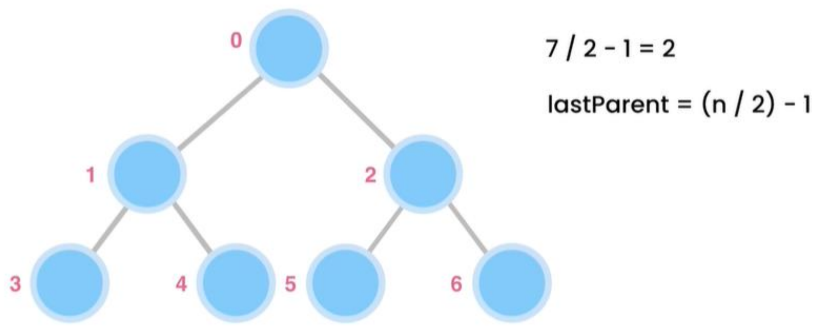
Input:

```
{5,3,8,4,1,2}
```

Output:

[8, 4, 5, 3, 1, 2]

V-(12)- Heapify (Optimized):-



If we continue our operation till last parent then we can lessens the recursion half because maximum nodes will be remaining in the leaf and we don't have to perform heapify for leaf nodes [***we only need to heapify parent nodes***] that's what the reason we should continue our recursion till lastparentIndex.

```
public static void heapifyOptimized(int []array){
    //half of recursion will decreased
    var lastParentIndex = array.length / 2 - 1;
    for(int i=lastParentIndex;i>=0;i--)
        heapify(array,i);
}
```

V-(13+14)- Kth largest Item:-

We have to find the kth largest item. Where our Algorithm using heap would be remove k-1 items from heap as our whole heap example using max heap & root node has the highest value. So, whenever we remove a value highest value will remove that's how we keep kth highest value and remove k-1 highest values. Then just return the value that exist in the root cause it is the required highest value.

```
public static int kthLargestItem(int [] array,int k){
//it is in the maxheap
    if(k<1 || k>array.length)
        throw new IllegalArgumentException();
    var heap = new Heap();
    for(var number:array)//inserting all the items in the array
        heap.insert(number);
    //now we have to delete the items till k-1 times
    // so that we got kth largest from root
    for(var i=0;i<k-1;i++){
        heap.remove();
    }
    return heap.max();
}
```

```
public int max(){//this method is in heap class
    if(isEmpty())
        throw new IllegalStateException();
    return items[0];
}
```

Exercises:-

1- Given an array of integers, check to see if this array represents a max heap.
Ans:-

```
public static boolean isMaxHeap(int[] array) {
    return isMaxHeap(array, 0);
}

private static boolean isMaxHeap(int[] array, int index) {
    // All leaf nodes are valid
    var lastParentIndex = (array.length - 2) / 2;
    //if we look at till last parent then it would be okay as per mosh
    if (index > lastParentIndex)
        return true;
    //find left and right child of every nodes till last parent
    // then compare them with the max heap property parent>=child
    var leftChildIndex = index * 2 + 1;
    var rightChildIndex = index * 2 + 2;

    var isValidParent =
            array[index] >= array[leftChildIndex] &&
                    array[index] >= array[rightChildIndex];

    return isValidParent &&
            isMaxHeap(array, leftChildIndex) &&
            isMaxHeap(array, rightChildIndex);
}
```

```
//test cases for max-heap checker
System.out.println(new Heap().isMaxHeap(new int[]{15,13,18,14,11,12}));//false
System.out.println(new Heap().isMaxHeap(new int[]{5,4,3,2,1}));//true
```

2- Implement a min heap. In this implementation, store the items in an array of nodes. Each node should have two fields: key (integer) and value (string). Nodes should be heapified based on their keys.
Ans:-

```
public class MinHeap {
    private class Node{
        private int key;
        private String value;
        public Node(int key,String value){
            this.key=key;
            this.value=value;
        }
    }
    private Node[] nodes = new Node[10];
    private int size;
    public void insert(int key,String value){
        if(isFull())
            throw new IllegalStateException();
        nodes[size++]=new Node(key,value);
        bubbleup();
    }
    private void bubbleup(){
        var index=size-1;
        while (index>0 && nodes[index].key<nodes[parent(index)].key){
            swap(index,parent(index));
            index=parent(index);
        }
    }
    private int parent(int index){
        return (index-1)/2;
    }
    private void swap(int first,int second){
        var temp = nodes[first];
        nodes[first] = nodes[second];
        nodes[second] = temp;
    }
    public boolean isFull(){
        return size== nodes.length;
    }
    public String remove(){
        if(isEmpty())
            throw new IllegalStateException();
        var root = nodes[0].value;
        nodes[0]=nodes[--size];

        bubbleDown();

        return root;
    }

    private void bubbleDown(){
        var index=0;
        while(index<=size && !isValidParent(index)){
            var largerChildIndex=smallerChildIndex(index);
            swap(index,largerChildIndex);
            index = largerChildIndex;
        }
```

```
    }

    private int smallerChildIndex(int index){
        //in max heap the largest value will be on the root but here we work
with min heap
        //that means lowest value should be at the root
        //For this reason, we find the smallest child here and swap between
smaller child and larger child
        //largerchild will go down ,for that we use bubble down method
        if(!hasLeftChild(index))
            return index;
        if(!hasRightChild(index))
            return leftChildIndex(index);
        return (leftChild(index).key<rightChild(index).key) ?
leftChildIndex(index):rightChildIndex(index);
    }

    private boolean hasLeftChild(int index) {
        return leftChildIndex(index) <= size;
    }

    private boolean hasRightChild(int index) {
        return rightChildIndex(index) <= size;
    }

    private boolean isValidParent(int index) {
        if (!hasLeftChild(index))
            return true;

        var isValid = nodes[index].key <= leftChild(index).key;

        if (hasRightChild(index))
            isValid &= nodes[index].key <= rightChild(index).key;

        return isValid;
    }

    private Node rightChild(int index) {
        return nodes[rightChildIndex(index)];
    }

    private Node leftChild(int index) {
        return nodes[leftChildIndex(index)];
    }

    private int leftChildIndex(int index) {
        return index * 2 + 1;
    }

    private int rightChildIndex(int index) {
        return index * 2 + 2;
    }

    public boolean isEmpty(){
        return size==0;
    }
}
```

3- Implement a min priority queue with the following operations:
• add(String value, int priority)
• remove()
• isEmpty()
Use the MinHeap class that you created in the last exercise. Items with a smaller priority should be moved to the beginning of the queue. Hint: use the priority of each item as the key in your min heap.

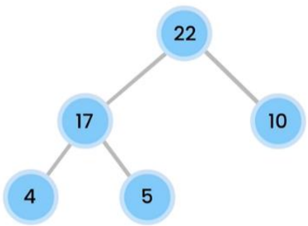
```
public class MinPriorityQueue {
    private MinHeap heap = new MinHeap();

    public void add(String value, int priority) {
        heap.insert(priority, value);
    }

    public String remove() {
        return heap.remove();
    }

    public boolean isEmpty() {
        return heap.isEmpty();
    }
}
```

SUMMARY:-



Heap is a complete binary tree where each level except the last level is filled from left to right. Heaps are two types . 1. Max heap 2. Min heap In Max heap parent>=child and in Min heap parent<=child. Adding or removing an items take **O(log n).** Cause it has to ***travel height of the tree*** at most. Heap *doesn't support look up values* cause of the nodes structure. It allow us to inspect the maximum or minimum value by just root node value. Getting root node value min or max heap identification is **O(1).**